

# Lock-free parallel garbage collection

Gao, H.; Groote, J.F.; Hesselink, W.H.

*Published in:*
PARALLEL AND DISTRIBUTED PROCESSING AND APPLICATIONS



*Publication date:*
2005

# Lock-Free Parallel Garbage Collection

H. Gao[1], J.F. Groote[2], and W.H. Hesselink[1]

[1] University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands
[2] Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

**Abstract.** This paper presents a lock-free parallel algorithm for *garbage collection* in a realistic model using synchronization primitives offered by machine architectures. *Mutators* and *collectors* can simultaneously operate on the data structure. In particular no strict alternation between usage and cleaning up is necessary, contrary to what is common in most other garbage collection algorithms.

We first design and prove an algorithm with a coarse grain of atomicity and subsequently apply the reduction theorem developed in [11] to implement the higher-level atomic steps by means of the low-level primitives.

## 1 Introduction

A *lock-free* (also called *non-blocking*) implementation of a shared object guarantees that within a finite number of steps always some process trying to perform an operation on the object will complete its task, independently of the activity and speed of other processes [12]. Since lock-free synchronizations are built without locks, they do not suffer from performance bottlenecks, which are often caused by locks and which can easily have a performance degrading effect of several orders of magnitude. In addition, lock-free synchronizations can offer progress guarantees. A number of researchers [1, 3, 12, 18] have proposed techniques for designing lock-free implementations. Essential for such implementations are advanced machine instructions such as *compare-and-swap* (*CAS*), or *load-linked* (*LL*)/*store-conditional* (*SC*).

In this paper we propose a lock-free implementation of mark&sweep *garbage collection* (GC). Garbage *collector*s are employed to identify at run-time which objects are no longer referenced by the *mutator*s (i.e. user programs). The heap space occupied by these objects is said to be *garbage* and must be re-cycled for subsequent new objects. The garbage collectors reclaim all garbage by adding them to a so called *free-list*, which keeps track of free memory.

There are several basic strategies for GC: reference counting, mark&sweep and copying. Reference counting algorithms can do their job incrementally (resulting in shorter collection pauses), but impose overhead on the mutators and fail to reclaim circular garbage. Mark&sweep algorithms can reclaim circular structures, and don't place any burden on the mutators like reference counting algorithms do, but tend to leave the heap fragmented. Copying algorithms can reduce fragmentation, but add the cost of copying data from one space to another

Y. Pan et al. (Eds.): ISPA 2005, LNCS 3758, pp. 263–274, 2005.

and require twice as much memory as a mark&sweep collector. Moreover, copying also requires that the programming language restrict address manipulation operations, which isn't true for C or C++.

One often encounters GC algorithms (e.g. [7, 8]) that employ stop-the-world mechanisms, which suspend all normal running threads and then perform GC. Such an algorithm introduces a global synchronization point between all threads and tends to become a scaling bottleneck that limits program performance and processor utilization. It is unacceptable when the system must guarantee response time of interactive applications. Therefore, to achieve parallel speed-ups on shared-memory multiprocessors, lock-free algorithms are of interest [17, 21].

There are several lock-free GC algorithms in the literature. The first one is due to Herlihy and Moss [13]. They present a lock-free copying GC algorithm, which uses excessive copying for moving objects to avoid blocking synchronization. In their algorithm, the failure of a participating thread can indefinitely prevent the freeing of unbounded memory. In [15], Hesselink and Groote give a wait-free (wait-freedom is stronger than lock-freedom) GC algorithm using reference counting. However, this collector applies only to a restricted programming model, in which objects are not allowed to be modified between creation and deletion, and is therefore generally limited. Detlefs et. al. [5] provide a lock-free GC algorithm using reference counting. The approach relies on a strong hardware primitive, namely *double-compare-and-swap* (*DCAS*) for atomic update of two distinct words in memory. Michael [20] presents an efficient lock-free memory management algorithm that does not require special operating system or hardware support. However, his algorithm only guarantees an upper bound on the number of removed nodes not yet freed at any time. This is undesirable because a single garbage node might use a large amount of resources and might never be reclaimed.

Mark&sweep algorithms do not move objects. They can thus coexist well with C/C++ code, where one never dares to move an object because of possible address computations, and are gaining popularity. Our lock-free mark&sweep algorithm is non-intrusive and features high-performance and reliability. Moreover, unlike most previously published Mark&sweep algorithms [2, 6, 7], we make no assumption on the maximum numbers of mutators and collectors that can operate concurrently. As far as we could find, no similar algorithm exist.

The correctness properties of any concurrent implementation are seldom easy to verify. This is in general even harder for lock-free algorithms. Our previous work [9] shows that providing correctness proofs for such algorithms require huge amounts of effort, time, and skill. In [11], we have developed a reduction theorem that enables us to reason about a lock-free program to be designed on a higher level than the synchronization primitives. Using the reduction theorem, fewer invariants are required and some invariants are easier to discover and formulate without considering the internal structure of the final implementation.

## 2 Specification

We assume a fixed set `Node` of nodes (cf. Fig. 1), each of which is identified with a unique label between 1 and $N$ for some $N \in \mathbb{N}$. The nodes in the set `free` are the

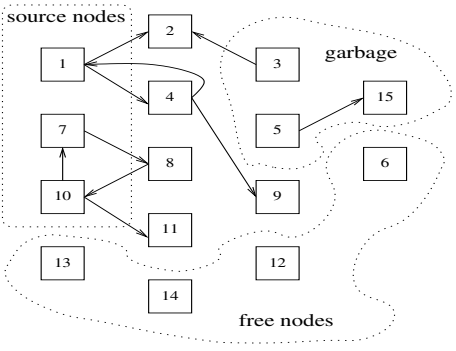


**Fig. 1.** A graph representation of the memory

*free nodes*. We model the heap as a finite directed graph of varying structure with a set of non-free nodes. Each node in the graph points to zero or more children (nodes), and the descendent relation may be circular. In the following context, we regard the attributes of nodes as arrays indexed by $1 \ldots N$. The number of children of a node $x$ is indicated by its `arity`, which is denoted by $\texttt{arity}[x]$. We let $C$ be the upper bound of the arities of the nodes. The expression $\texttt{child}[x, j]$ stands for the pointer to the $j$th child of node $x$, where $1 \leq j \leq \texttt{arity}[x]$.

A node is called a *root* when some process has direct read access to it. Each application process $p$ maintains a private set $roots_p$ that holds its root nodes. The set *Roots* is the union of all $roots_p$ for all processes $p$.

Access to nodes can be transferred between processes. We assume that there is a two-dimensional array `Mbox` indexed with a pair of processes that serves as mailboxes. If process $p$ allows process $q$ to access some node $x$, it writes $x$ at $\texttt{Mbox}[p, q]$ using *Send*. Then, process $q$ can claim the access by calling *Receive*.

We call a node a *source node* if the node is either in *Roots* or in some mailbox. A node is called *accessible* iff it is reachable by following a chain of pointers from a source node. Free nodes must not be accessible. Only nodes in the `free` set are allowed to be allocated by the mutators. A node is said to be a *garbage node* if it is neither accessible nor in the `free` set. Garbage collectors compute the set of nodes reachable from a set of source nodes and reclaim all garbage nodes by placing them into the `free` set. More formally, we define

$$\begin{aligned} R(p,\ x) \ &\equiv\ (\exists z \in roots_p\colon z \xrightarrow{*} x), \\ R(x) \ &\equiv\ (\exists z \in Roots\colon z \xrightarrow{*} x)\ \vee \\ &\qquad (\exists p,\ q \in \texttt{Process}\colon \texttt{Mbox}[p,\ q] \xrightarrow{*} x), \end{aligned}$$

where the reachability relation $\xrightarrow{*}$ is the reflexive transitive closure of relation $\rightarrow$ on nodes defined by: $z \rightarrow x \ \equiv\ (\exists k\colon 1 \ldots \texttt{arity}[z]\colon \texttt{child}[z,\ k] = x)$. The fact that a node $x$ is a garbage node is formalized by: $\neg R(x) \wedge x \notin \texttt{free}$.

The interface of the mutators consists of a shared data structure of nodes, and a number of procedures that can be called in the application processes. We assume there are in total $P$ concurrently executing sequential processes. In the

text of the procedures specified as follows, we use *me* to stand for the process that invokes the procedure. We use angular brackets $\langle\ \rangle$ to indicate that embraced statements are (thought to be) executed atomically.

**proc** *Create*(): Node
  **local** $x$ : Node;
  $\langle$ **when** available **extract** $x$ **from** free;
    arity$[x] := 0$; $roots_{me} := roots_{me} \cup \{x\}$; $\rangle$
  **return** $x$;
**proc** *AddChild*($x$, $y$: Node): Bool
{ $R(me, x) \wedge R(me, y)$ }
  **local** *suc* : Bool;
  $\langle$ $suc :=$ (arity$[x] < C$);
    **if** *suc* **then** arity$[x]$++; child$[x,$ arity$[x]] := y$; **fi** $\rangle$
  **return** *suc*;
**proc** *GetChild*($x$: Node, *rth*: $\mathbb{N}$): Node $\cup \{0\}$
{ $R(me, x)$ }
  **local** $y$ : Node $\cup \{0\}$;
  $\langle$ **if** $1 \leq rth \leq$ arity$[x]$ **then** $y :=$ child$[x, rth]$; **else** $y := 0$; **fi** $\rangle$
  **return** $y$;
**proc** *Make*($c$: **array** [ ] **of** Node, $n$: $1 \ldots C$): Node
{ $\forall j$: $1 \leq j \leq n$: $R(me, c[j])$ }
  **local** $x$ : Node; $j$ : $\mathbb{N}$;
  $\langle$ **when** available **extract** $x$ **from** free;
    **for** $j := 1$ **to** $n$ **do** child$[x, j] := c[j]$ **od**;
    arity$[x] := n$; $roots_{me} := roots_{me} \cup \{x\}$; $\rangle$
  **return** $x$;
**proc** *Protect*($x$: Node)
{ $R(me, x) \wedge x \notin roots_{me}$ }
  $\langle$ $roots_{me} := roots_{me} \cup \{x\}$; $\rangle$
  **return**;
**proc** *UnProtect*($z$: Node)
{ $z \in roots_{me}$ }
  $\langle$ $roots_{me} := roots_{me} \setminus \{z\}$; $\rangle$
  **return**;
**proc** *Send*($x$: Node, $r$: Process)
{ $R(me, x) \wedge$ Mbox$[me, r] = 0$ }
  $\langle$ Mbox$[me, r] := x$; $\rangle$
  **return**;
**proc** *Receive*($r$: Process): Node
{ Mbox$[r, me] \neq 0$ }
  **local** $x$ : Node;
  $\langle$ $x :=$ Mbox$[r, me]$;
    Mbox$[r, me] := 0$; $roots_{me} := roots_{me} \cup \{x\}$; $\rangle$
  **return** $x$;

The application programmers are responsible for ensuring that an offered procedure is called only when its precondition (enclosed by braces { } if there is any) holds. The condition "available" in *Create* and *Make* is implementation dependent. When an allocation request cannot be met from the free memory,

the mutator either waits, or invokes a new round of GC to free more garbage. The threshold value that determines whether or not to invoke a new round of GC can be customized by the user.

Behind this abstract "user system" there is a collection of garbage collecting processes. A garbage collector does not modify the memory graph but only manipulate the `free` set. To specify that GC does happen and is eventually exhaustive, we give the liveness property, i.e. every garbage node will be eventually put into the `free` set by a garbage collector.

## 3 A Higher-Level Implementation

The idea behind most GC algorithms in use is to first recursively trace all reachable nodes starting from root nodes, then nodes not reached are considered garbage and can be collected. We present a lock-free implementation that comes close to the classical mark&sweep algorithms.

We first extend the specification to a high-level implementation, where all actions on shared variables are separated into distinct atomic accesses except for some special commands enclosed by angular brackets $\langle \ldots \rangle$. In order to be able to finally transform the higher-level algorithm into the low-level algorithm using our reduction theorem developed in [11], we require that every labeled atomic group of statements in the higher-level algorithm refer to at most one shared node.

### 3.1 Data Structure

The data structure we use in the higher-level implementation is shown in Fig. 2. Besides fields `arity` and `child`, each node has one of three colors: *white*, *black* and *grey*. All *black* nodes reachable from a source node are interpreted as accessible nodes, and all other *black* nodes are garbage. *Grey* is a transient color that only occurs during GC. The `free` set is implemented as a virtual set that contains all *white* nodes.

Since any accessible node must not be freed as garbage, the system needs to keep track of source nodes that are created by a process and may still be referred to by other processes. We introduce a field `srcnt` for each node to count all references (processes and mailboxes) to the node as a source node.

To avoid possible interference between mutators and collectors, the updates of the field `srcnt` of the node, upon deletion from the *roots* set, is postponed. We use the field `freecnt` to count the postponed decrementings of `srcnt`. The fields `ari` and `father` record the number of children a node has at the beginning of GC and the parent node of a node in a tree traversed from a source node by collectors, respectively.

We use a shared variable `shRnd` to hold the round number of the current GC, together with an additional field `round` in the record of a node. The private variable *rnd* is a private copy of the shared variable `shRnd`. The global private variable *toBeC* is used to transfer information about checked nodes between

**Constant**
$P =$ `number of processes`; $N =$ `number of nodes`;
$C =$ `upper bound of number of children`;
**Type**
`colorType`: {*white, black, grey*};
`nodeType`: **record** =
`arity, srcnt, freecnt, ari, round`: $\mathbb{N}$;
`child`: **array** $[1 \dots C]$ **of** $1 \dots N$;
`color`: `colorType`; `father`: $\mathbb{N} \cup \{-1\}$;
**end**
**Shared variables**
`Mbox`: **array** $[1 \dots P,\ 1 \dots P]$ **of** $0 \dots N$;
`Node`: **array** $[1 \dots N]$ **of** `nodeType`; `shRnd`: $\mathbb{N}$;
**Private variables**
*roots*, *toBeC*: **a subset of** $1 \dots N$; *rnd*: $\mathbb{N}$;
**Initialization**:
`shRnd` $= 1 \wedge \forall x$: $1 \dots N$: `round`$[x] = 1$;

**Fig. 2.** Data Structure

internal calls. There is also a local private variable *toBeD* for representing the set of source nodes to be tracked from.

### 3.2 Algorithm

In this section, we give a higher-level implementation for the collectors and the mutators. Since the same sequential program can be executed by all processes, we adopt the convention that every private variable name can be subscripted by the process identifier. In particular, $pc_p$ is the program counter of process $p$. We do not write `Node`$[x].f$ but $f[x]$. We denote `color`$[x] =$ *white* by *white*$(x)$, and similarly for the other two colors. Brackets $[\![\ ]\!]$ and the actions between parenthesis $(\!|\ |\!)$ can be ignored in the implementation. They only serve in the proof of correctness. We will explain this in section 4.

**Collectors.** Our garbage collectors are encoded in the procedure *GCollect* as shown in Fig. 3. It consists of three phases: (1) initialization: paint all *black* nodes *grey*, (2) marking: paint all *grey* nodes reachable from the source nodes back to *black* after traversing the memory graph, and (3) sweeping: reclaim all garbage by painting all remaining *grey* nodes *white*.

In the first phase, the processes only need to paint the *black* nodes *grey* since the *white* nodes can not be garbage. Moreover, we let the field `father` of each node with positive `srcnt` be 0, and that of other nodes be $-1$. As the algorithm allows parallel use of mutators, being a source node is not stable. For simplicity, we call a node $x$ with `father`$[x] = 0$ an *old source node*.

In line 108, a delayed initialization on node $x$ will be skipped since `round`$[x]$ is never decreased. As usual with version numbers, here we assume that sufficient

```
proc GCollect() =
      local x: 1...N; toBeD: a subset of 1...N;
100:  rnd := shRnd; toBeC := {1,...,N};
101:  while shRnd = rnd ∧ toBeC ≠ ∅ do
          choose x ∈ toBeC;
108:      ⟨ if round[x] = rnd then
              round[x] := rnd + 1; ari[x] := arity[x];
              if black(x) then color[x] := grey; fi;
              if srcnt[x] > 0 then father[x] := 0; else father[x] := −1; fi; fi; ⟩
          toBeC := toBeC \ {x}; od;
121:  toBeC := {1,...,N}; toBeD := {1,...,N};
122:  while shRnd = rnd ∧ toBeD ≠ ∅ do
          choose x ∈ toBeD;
126:      toBeD := toBeD \ {x};
          ⟨ if father[x] = 0 then Mark_stack(x); fi; ⟩ od;
129:  while shRnd = rnd ∧ toBeC ≠ ∅ do
          choose x ∈ toBeC;
134:      ⟨ if round[x] = rnd + 1 ∧ grey(x) then
                color[x] := white;
                (| assert ¬R(x) ∧ x ∉ free; free := free ∪ x; |) fi; ⟩
          toBeC := toBeC \ {x}; od;
135:  ⟨ if rnd = shRnd then shRnd := rnd + 1; fi; ⟩
137:  return
end GCollect.
```

**Fig. 3.** Procedure *GCollect*

bits are allocated for the version numbers to ensure that they cannot "wrap around" during the interval of a process's GC cycle.

In the second phase, lines 121-126, the processes build a forest in the set of all reachable nodes starting from the old source nodes. Trees in the forest are mutually disjoint. Each of them is rooted by a chosen old source node, and is established via calling *Mark_stack* (see Fig. 4) in a *while* loop. During *Mark_stack*, all the *grey* nodes on the tree are painted *black* in the order from the leaf to the root.

The procedure *Mark_stack* is mainly a form of graph search, and it was initially designed as a recursive procedure. Since we want to prove the correctness of our algorithm with PVS, we eliminated the recursion in favor of an explicit stack. The private variable *toBeC* serves to ensure that the search of a collector traverses every node at most once. This is important since the memory graph may have cycles and nodes may be reachable from different old source nodes.

In *Mark_stack*, lines 151-163, the tree is established by setting the `father` pointers. Since the memory graph may have cycles, the processes must reach consensus about the tree. The processes starting from the same old source node cooperate with each other, and are in competition with others to expand the tree to all nodes reached.

```
proc Mark_stack(x: 1...N) =
    local w, y: 1...N; suc: Bool; j, k: ℕ;
    stack: Stack; head: ℕ; set: a subset of 1...N;
    ch: [1...C] of 1...N;
150: toBeC := toBeC \ {x}; set := {x}; head := 0;
151: while shRnd = rnd ∧ set ≠ ∅ do
157:     choose w ∈ set; set := set \ {w};
         ⟨ if grey(w) ∧ round[w] = rnd + 1 then
             k := ari[w];
             for j := 1 to k do ch[j] := child[w, j] od; ⟩
             head++; stack[head] := w; j := 1;
158:         while shRnd = rnd ∧ j ≤ k do
                 y := ch[j];
                 if y ∉ toBeC then j++;
                 else
163:                ⟨ if father[y] ∈ {−1, w} ∧ grey(y)
                           ∧round[y] = rnd + 1 then
                       father[y] := w; ⟩ set := set ∪ {y};
                       toBeC := toBeC \ {y}; fi;
                    j++; fi; od; fi; od;
168: while shRnd = rnd ∧ head ≠ 0 do
175:     y := stack[head]; head--;
         ⟨ if grey(y) ∧ round[y] = rnd + 1 then
             srcnt[x] := srcnt[x] − freecnt[x];
             color[y] := black; freecnt[x] := 0; fi; ⟩ od;
180: return
end Mark_stack.
```

**Fig. 4.** Procedure *Mark_stack*

The order for choosing an element from the local variable *set* is irrelevant for correctness, but relevant for efficiency. The search is a depth first search if the order is first in last out. The search is a breadth first search if the order is first in first out. Starting from the chosen old source node, all nodes on the tree are pushed onto the local stack after their children have been stored. The order of the elements pushed onto the stack is essential for correctness.

After the tree has been established, the process paints all *grey* nodes *black* in the order in which they are popped from the *stack* (lines 168-175). When a node in the tree is painted *black*, its descendants (with respect to the *father* relation) in the tree must have been painted *black* already. So the other processes need not trace or paint the subtree starting from that node. At the end of *Mark_stack*, the process returns to the procedure *GCollect* to traverse another tree from another old source node.

In the third phase, lines 129-134, processes try to re-cycle all remaining *grey* nodes by coloring them *white* (i.e. adding them to the `free` set). The main proof obligation for the algorithm is that all nodes being freed are not accessible. When the fastest process executes line 135, the shared variable `shRnd` is incremented to notify all other collectors that this round of GC is completed.

**Mutators.** The higher-level implementations of the procedures for the mutators are relatively easy. For reasons of space, in Fig. 5 we only provide the code for procedure *Make* (see [10] for the remainning). In the code, "time to do GC" indicates that some variable, like time or the amount of free memory, reaches a threshold value.

```
proc Make(c: array [ ] of 1...N, n: 1...C): 1...N =
{ ∀ j: 1...n: R(me, c[j]) }
      local x: 1...N; j: ℕ;
      while true do
300:     choose x ∈ [1...N];
306:     ⟨ if white(x) then
             color[x] := black; srcnt[x] := 1;
             (| assert x ∈ free; free := free \ x; |)
             [[ for j := 1 to n do child[x, j] := c[j]; od
               arity[x] := n; roots := roots ∪ {x}; ]] ⟩
             break;
308:     elseif time to do GC then GCollect(); fi; od;
310:  [[ return x ]]
end Make.
```

**Fig. 5.** Procedure *Make*

## 4 Correctness

The main issue of the algorithm is how to ensure the correct execution of collectors and mutators when they concurrently compete with each other for the same data structure. The algorithm is correct if it behaves properly for all interleavings. Here we only give a sketch of the correctness of the algorithm. For the complete mechanical proof, we refer to [14].

We need to distinguish safety properties and liveness properties. The main aspect of safety is functional correctness and atomicity, say in the sense of [19]. We prove partial correctness of the implementation by showing that each procedure of the implementation executes its specification command exactly once and that the resulting value of the implementation equals the resulting value in the specification. As shown in Fig. 3 to Fig. 5, we extend the implementations with auxiliary variables and commands used in the specification. For simplicity, we use brackets [[ ]] to enclose the specification commands that perform the same actions as the implementation, and parenthesis (| |) to enclose the specification commands that can be deleted in the implementation.

GC is an internal affair not relevant for the users of the routines. *GCollect* cannot be invoked explicitly, but will only be invoked implicitly in, e.g. *Make*. This means we only need to prove the match of the specifications and implementations for all user programs, but not for *GCollect*. Instead, the main safety property we have proved for *GCollect* is that the system only collects garbage,

i.e. that an accessible node is never freed. This is expressed in the invariant *I1*: $white(x) \Rightarrow \neg R(x)$.

Furthermore, we also need to prove that all preconditions of the interface procedures are stable under the actions of the other processes. Process $p$ can ensure its rights to have access to node $x$ by checking the predicate $R(p, x)$, independently.

A liveness property asserts that program execution eventually reaches some desirable state. In our case, we want to ensure it is always the case that every garbage node is eventually collected. That is, $\neg R(x) \leadsto white(x)$, where $\leadsto$ is the "*leads-to*" relation defined by: $(P \leadsto Q) \equiv \Box(P \Rightarrow \Diamond Q)$.

We actually prove something stronger, viz., that, every inaccessible node is painted *white* within two rounds of GC.

**Theorem 1.** *For any integer* $m$,
$\mathtt{shRnd} = m \wedge \neg R(x) \quad \leadsto \quad \mathtt{shRnd} \leq m + 2 \wedge white(x)$.

## 5 The Low-Level Implementation

Synchronization primitives *LL* and *SC*, proposed by Jensen et. al. [16], have found widespread acceptance in modern processor architectures (e.g. MIPS II, PowerPC and Alpha architectures). These instructions are closely related to the *CAS*, and together implement an atomic Read/Write cycle.

At the cost of copying an object's data before an operation, Herlihy [12] introduced a general methodology to transfer a sequential implementation of any data structure into a lock-free synchronization by means of synchronization primitives *LL* and *SC*.

In [11], we formalize Herlihy's methodology [12] and develop a reduction theorem that enables us to reason about a general lock-free algorithm to be designed on a higher level than the synchronization primitives. A reduction theorem is a general rule for deriving an "equivalent" higher-level specification from a lower-level one in some suitable sense [4]. The big advantage is that substantial pieces of the concrete program can be dealt with as atomic statements on the higher level and thus the correctness can be more easily verified.

In the higher-level implementation (from Fig. 3 to Fig. 5), instruction 135 is simply a *CAS* instruction offered by machine architectures. Each of all other special commands enclosed by angular brackets $\langle \ldots \rangle$ only refer one shared node and some private variables, and therefore can be transformed into low-level lock-free implementations using our reduction theorem. The transformation is straightforward, and we refer the reader to [14].

## 6 Conclusions

We present a lock-free parallel algorithm for mark&sweep GC in a realistic model by means of synchronization primitives *compare-and-swap* (*CAS*) and *load-linked* (*LL*)/*store-conditional* (*SC*) offered by machine architectures. Our

algorithm allows to collect a circular data structure and makes no assumption on the maximum number of mutators and collectors that can operate concurrently during GC. The efficiency of GC can be enhanced when more processors are involved in it.

Formal verification is desirable because there could be subtle bugs as the complexity of algorithms increases. To ensure our correctness proof presented in the paper is not flawed, we use the higher-order interactive theorem prover PVS for mechanical support. For the complete mechanical proof, we refer the reader to [14].

In the interface we did not provide a procedure for deleting a child of a node. However, this extension is rather straightforward after we have done the following two steps. First, introduce an additional field of a boolean array in the record of a node to record whether a child of a node is deleted. The boolean array should restrict only the mutators not the collectors from accessing a "deleted" child via the pointers of children. Secondly, similarly to what we did with unprotecting a source node, we need to modify line 175 to let the deletions of some "deleted" children be really operated. Since we don't think deleting a child is a main operation of GC, we didn't incorporate it. However, the correctness of this extension should not be difficult to verify.

The entrenched problem inherited from classical mark&sweep algorithms is that our algorithm may also result in severe memory fragmentation, with lots of small blocks. It is possible that there will be no block of memory on the free list large enough to hold a large object, such as an array. Thus, it is important to move free blocks that happen to be adjacent in memory. We plan in the future to incorporate some appropriate copying technique in our algorithm.

## References


1. G. Barnes. A method for implementing lock-free data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993.
2. M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on programming Languages and Systems*, 6(3):333–344, 1984.
3. B.N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 264–274, 1993.
4. E. Cohen and L. Lamport. Reduction in TLA. In *Proceedings of the 9th International Conference on Concurrency Theory*, pages 317–331, 1998.
5. D.L. Detlefs, P.A. Martin, M. Moir, and G.L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–71, December 2002.
6. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.
7. T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–14. ACM Press, 1997.

8. C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.
9. H. Gao, J.F. Groote, and W.H. Hesselink. Almost wait-free resizable hashtables (extended abstract). In *Proceedings of 18th International Parallel & Distributed Processing Symposium (IPDPS)*, April 2004.
10. H. Gao, J.F. Groote, and W.H. Hesselink. Lock-free parallel garbage collection by mark&sweep. Technical Report CS-Report CSR-04-31, Eindhoven University of Technology, The Netherlands, 2004.
11. H. Gao and W.H. Hesselink. A formal reduction for lock-free parallel algorithms. In *Proceedings of the 16th conference on Computer Aided Verification (CAV)*, July 2004.
12. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
13. M.P. Herlihy and J.E.B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, 1992.
14. W.H. Hesselink. `http://www.cs.rug.nl/~wim/mechver /garbage_collection`
15. W.H. Hesselink and J.F. Groote. Wait-free concurrent memory management by Create, and Read until Deletion. *Distributed Computing*, 14(1):31–39, January 2001.
16. E.H. Jensen, G.W. Hagensen, and J.M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livemore National Laboratory, January 1987.
17. P.C. Kanellakis and A. A. Shvartsman. *Fault-Tolerant Parallel Computation.* Kluwer Academic Publishers, 1997.
18. V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 314–323. ACM Press, 2003.
19. N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.
20. M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30. ACM Press, 2002.
21. H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM Symposium on Applied computing*, pages 1438–1445, 2004.